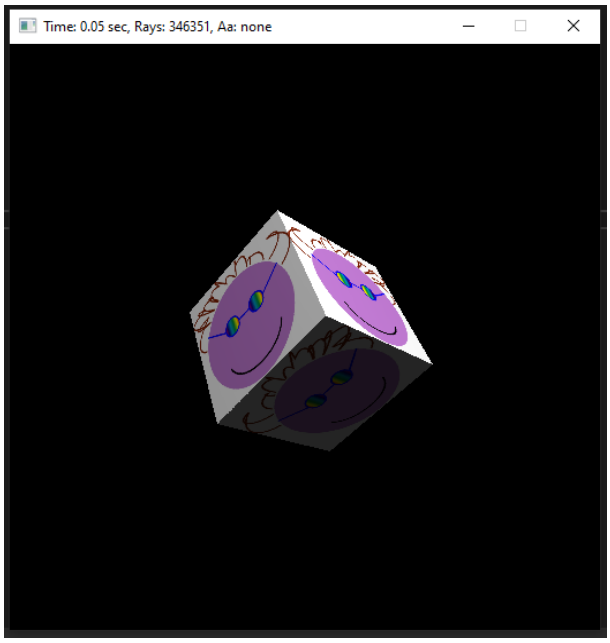
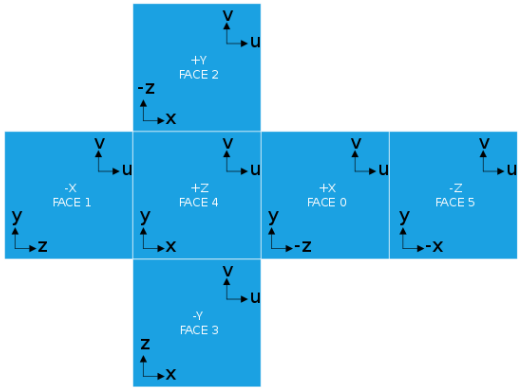
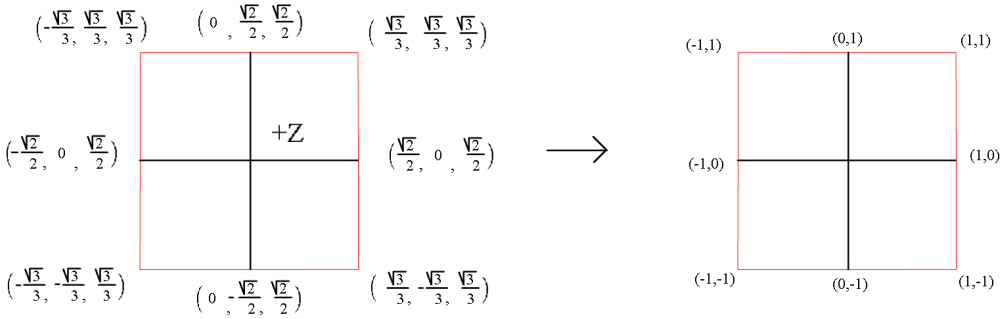
Matthew Jagen
2/15/2023

I started this assignment by implementing texture mapping and cube mapping. Here's some examples of my finished texture and cube mapping:



For cube mapping, I followed these handy graphics shared in the assignment 2 discord for how to orient the images and calculate which textel to grab from the cubemap's texture maps





Here's a quick snippet of the CubeMap::getColor() code:

```
glm::dvec3 CubeMap::getColor(ray r) const
{
    // YOUR CODE HERE
    // FIXME: Implement Cube Map here

    //determine which texture in the cube map we should be looking at
    int t_idx = 0;
    glm::dvec3 dir = r.getDirection();
    double dir_comp = 0.0;
    double u_comp = 0.0;
    double v_comp = 0.0;
    if (abs(dir[0]) >= abs(dir[1]) && abs(dir[0]) >= abs(dir[2])) {
        //this vector is oriented most strongly in the X direction
        if (dir[0] >= 0) {  //look at the positive X image (right)
            t_idx = 0;
            dir_comp = dir[0];
            u_comp = -dir[2]; //flip the sign because negative Z values should be left-of-center
            v_comp = dir[1];
        }
        else {              //look at the negative X image (left)
            t_idx = 1;
            dir_comp = -dir[0];
            u_comp = dir[2];
            v_comp = dir[1];
        }
    }
```

The other code for Y and Z are nearly identical. Then, we just convert our u and v components to actual coordinates and grab them from the texture map at t_idx.

```
    //calculate where on the texture map we need to sample
    double u_coord = u_comp / dir_comp; //now ranges from [-1, 1]
    u_coord = (u_coord + 1) / 2.0; //now ranges from [0, 1]
    double v_coord = v_comp / dir_comp;
    v_coord = (v_coord + 1) / 2.0;

    return tMap[t_idx]->getMappedValue(glm::dvec2(u_coord, v_coord));
}
```

Then was antialiasing, which was rather simple. We just check to see if the aa switch is on and if so, partition the pixel into supersamples and then combine the samples at the end

```
if (traceUI->aaSwitch()) {
    int samples = traceUI->getSuperSamples();
    for (int n = 0; n < samples; n++) {
        for (int m = 0; m < samples; m++) {
            double offs = 1.0 / samples; //offset of one sample to another
            double b_offs = 0.5 / samples; //offset of all samples from the edge
            double x = ((double(i) + b_offs + offs * m)/ double(buffer_width));
            double y = ((double(j) + b_offs + offs * n) / double(buffer_width));
            col = col + trace(x, y);
        }
    }
    col = col / double((samples * samples));

}
else {
    // set ray cast offset to middle of pixel using + 0.5
    double x = (double(i) + 0.5) / double(buffer_width);
    double y = (double(j) + 0.5) / double(buffer_height);

    col = trace(x, y);
}
```

no more jaggies!

And finally, my KdTree implementation. I designed my KdTree<T> nodes to all have their own bounding box, SplitNodes to have an axis, position, left child, and right child, and the leaf nodes to just have an object_list vector.

For building the tree, the first piece of hard work is finding the split plane candidates. To do this, I extract all candidate planes for an axis and store them in a vector of tuple<int, double>s (the int in the tuple is the axis and the double is the position)

```
vector<tuple<int, double>> candidates;
glm::dvec3 bbmin = boundingbox.getMin();
glm::dvec3 bbmax = boundingbox.getMax();
for (int a = 0; a < 3; a++) {
    //loop through all axes
    for (int g = 0; g < obj_list.size(); ++g) {
        BoundingBox obj_bbox = obj_list[g]->getBoundingBox();
        double p1 = obj_bbox.getMin()[a];
        double p2 = obj_bbox.getMax()[a];
        if (p1 > bbmin[a] && p1 < bbmax[a])
            candidates.push_back(tuple<int, double>(a, p1));
        if (p2 > bbmin[a] && p2 < bbmax[a])
            candidates.push_back(tuple<int, double>(a, p2));
    }
}
double minSAM = 9999999.0;
tuple<int, double> bestSplit(-1, 0.0);
double bboxarea = boundingbox.area();
for (int p = 0; p < candidates.size(); ++p) {
    int split_axis = get<0>(candidates[p]);
    double split_part = get<1>(candidates[p]);
    tuple<int, int> lr = countObjects(obj_list, split_axis, split_part);
    double bblen = bbmax[split_axis] - bbmin[split_axis];
    double splitdist = split_part - bbmin[split_axis];
    double leftArea = bboxarea * (splitdist / bblen);
    double rightArea = bboxarea - leftArea;
    double SAM = get<0>(lr) * leftArea + get<1>(lr) * rightArea;
    if (get<0>(bestSplit) == -1 || SAM < minSAM) {
        minSAM = SAM;
        bestSplit = candidates[p];
    }
}
return bestSplit;
```

Then, I iterate through the list and calculate the split with the lowest surface area metric.

Back in the main buildTree function, the next step is to add all of this node's objects to the left and right object lists.

```
vector<Geometry*> left_obj_list;
vector<Geometry*> right_obj_list;
for (const auto& obj : obj_list) {
    BoundingBox obj_box = obj->getBoundingBox();
    double min = obj_box.getMin()[axis];
    double max = obj_box.getMax()[axis];
    if (min < pos)
        left_obj_list.push_back(obj);
    if (max > pos)
        right_obj_list.push_back(obj);
    if (min == pos && max == pos) {
        //surface lies on our split plane, check normal
        if (obj->getNormal()[axis] < 0)
            left_obj_list.push_back(obj);
        else
            right_obj_list.push_back(obj);
    }
}
if (left_obj_list.size() == 0 || right_obj_list.size() == 0) {
    return new LeafNode<T>(obj_list, boundingbox);
}
```

If either list is 0 we just decide to terminate into a leaf node instead of having a leaf with no objects.

And lastly, we do some work to prepare for recursion and then we build the left and right sides of this node.

```
glm::dvec3 bbmin = boundingbox.getMin();
glm::dvec3 bbmax = boundingbox.getMax();
glm::dvec3 lbbmin(bbmin[0], bbmin[1], bbmin[2]);
glm::dvec3 lbbmax(bbmax[0], bbmax[1], bbmax[2]);
lbbmax[axis] = pos;
glm::dvec3 rbbmin(bbmin[0], bbmin[1], bbmin[2]);
glm::dvec3 rbbmax(bbmax[0], bbmax[1], bbmax[2]);
rbbmin[axis] = pos;
return new SplitNode<T>(boundingbox, axis, pos,
    buildTree(left_obj_list, BoundingBox(lbbmin, lbbmax), depth, leafsize),
    buildTree(right_obj_list, BoundingBox(rbbmin, rbbmax), depth, leafsize));
```

For traversal through the KdTree, I use the BoundingBox that is related to every KdTree<T> to see if the ray intersects with this node. If it does and it's a split node, it will recurse into the left and right child. And if it's a leaf node, it will add the node's object list pointers to the candidate list.

Virtual function in class KdTree:

```
virtual bool getCandidates(ray, vector<Geometry*>*, double, double) = 0;
```

Implementation in SplitNode:

```
bool getCandidates(ray r, vector<Geometry*>* obj_list, double tmin, double tmax) {
    double i_tmin;
    double i_tmax;
    if (this->bbox.intersect(r, i_tmin, i_tmax)) {
        left->getCandidates(r, obj_list, tmin, tmax);
        right->getCandidates(r, obj_list, tmin, tmax);
        return true;
    }
    return false;
}
```

Implementation in LeafNode:

```
bool getCandidates(ray r, vector<Geometry*>* obj_list, double tmin, double tmax) {
    double i_tmin;
    double i_tmax;
    if (this->bbox.intersect(r, i_tmin, i_tmax)) {
        for (int i = 0; i < object_list.size(); i++) {
            obj_list->push_back(object_list[i]);
        }
        return true;
    }
    return false;
}
```

In the end, it seems that my render times are mostly bottlenecked by the time it takes to build the KdTree (unless using high anti-aliasing on dragon.ray).